How to find leaks in OPeNDAP SSFs

Use this procedure to find the source of memory leaks on OPeNDAP server-side code.

We will be using the valgrind tool with the leak checker. You may need root or administrative privileges to start/stop the OPeNDAP server. You might want to create a new OPeNDAP instance or ensure that others won't attempt to use the server during profiling.

Step-by-step guide

Run Valgrind to capture profile data

1. Setup environment:
    a. `hyrax_root% source spath.sh`
2. Stop OPeNDAP OLFS+BES (if currently running):
    a. `tomcat_root% ./bin/shutdown.sh`
    b. `hyrax_root% ./bin/besctl stop`
3. Start the BES using valgrind + leak checker. Trace children is required because server-side functions run in a forked executable called *be slistener*. Output will be captured in the file "log.txt" file.
    a. `tomcat_root% ./bin/startup.sh`
    b. `hyrax_root% valgrind --trace-children=yes --leak-check=full ./bin/besctl start  > log.txt 2>&1`
4. Run a test that hits this OPeNDAP server. Make sure to keep it on the small side since it will run much more slowly than normal. Note that the leak could be in the OPeNDAP loader also – for example we found a leak in the HDF EOS file. For that reason, the type of data file you use can have an impact.
5. Make note of what *beslistener* pid is running your SSF function. "Top" is usually plenty good for that.
6. Stop the server again, and restart it as normal if desired
    a. `tomcat_root% ./bin/shutdown.sh`
    b. `hyrax_root% ./bin/besctl stop`

Analyze the data

The log.txt files can be analyzed directly. There are sections for each pid, so find the pid of your *beslistener*. Pay special attention to the sections that say "definitely leaked" – these are leaks. Go to the stack trace section and valgrind will tell you precisely where the leaked memory was allocated.

Notes for OPeNDAP Developers

In normal C++ programming, anything allocated with **new** must be de-allocated with **delete**, or else a leak will occur.

The rules for libdap objects like Array, Structure, Vector, Str, etc. are a bit more complicated. You must delete **only** the objects you need to and not the ones you don't... otherwise your SSF will either leak or crash.

The general rules are:

- When another libdap object takes *ownership* of your object, you must **not** delete it.
- If your libdap object is used as a SSF return value you must delete it **only** when used as a nested SSF (i,.e. it is passed to another SSF), and is not the final return value.

Consider the following example.

- In function **test**, the Float64 object does not need to be deleted, because the Array "retVal" takes ownership of it.
- If the client invokes your SSF as *somefile.nc?test()* then all is well. *retVal* does not leak, even though you never deleted it.
- If the client invokes your SSF as in *somefile.nc?test2(test())* then *retVal* must be deleted in test2 or else it will leak.

```
void miicfunctions::function_test(int argc, libdap::BaseType * argv[],
libdap::DDS &dds, libdap::BaseType **btpp)
{
  Array* retVal = new Array("data",0);
  retVal->add_var_nocopy(new Float64("data"));

  *btpp = retVal;
}

void miicfunctions::function_test2(int argc, libdap::BaseType * argv[],
libdap::DDS &dds, libdap::BaseType **btpp)
{
  Array* array = dynamic_cast<libdap::Array*>(argv[0]);
  ...
  delete array;
}
```

Related articles

- How to setup & run a MIIC OPeNDAP Server
- How to build the MIIC OPeNDAP plugin
- How to debug OPeNDAP SSFs
- How to Profile OPeNDAP Server
- How to plot OPeNDAP requests per minute